Programming Challenge: Recursive List Processing and HOFs

Abstract: This assignment builds a library of small but useful functions that primarily use recursion to work properly. This solution document shows many uses of those functions alone and when used together.

Task 1:

Task 1 Code:

```
( defun singleton-p ( l )
   ( cond
      ( ( null l ) nil )
      ( ( null ( cdr l ) ) t )
      ( t nil )
   )
)

( defun rac ( l )
   ( cond
      ( ( singleton-p l )
         ( car l )
      )
      ( t
         ( rac ( cdr l ) )
      )
   )
)

( defun rdc ( l )
   ( cond
      ( ( singleton-p l )
         ()
      )
      ( t
         ( cons ( car l ) ( rdc ( cdr l ) ) )
      )
   )
)
```

```
( defun snoc ( o l )
   ( cond
      ( ( null l )
        ( list o )
      )
      ( t
        ( cons ( car l ) ( snoc o ( cdr l ) ) )
      )
   )
)

( defun palindrome-p ( l )
   ( cond
      ( ( null l )
        t
      )
      ( ( singleton-p l )
        t
      )
      ( ( equal ( car l ) ( rac l ) )
        ( palindrome-p ( cdr ( rdc l ) ) )
      )
      ( t
        nil
      )
    )
)
```

Task 1 Demo:

```
[2]> ( singleton-p '( a ) )
T
[3]> ( singleton-p '( a b ) )
NIL
[4]> ( singleton-p '( a b c d e f g ) )
NIL
[5]> ( trace rac )
;; Tracing function RAC.
(RAC)
[6]> ( rac '( a ) )
1. Trace: (RAC '(A))
1. Trace: RAC ==> A
A
[7]> ( rac '( a b c d ) )
1. Trace: (RAC '(A B C D))
2. Trace: (RAC '(B C D))
3. Trace: (RAC '(C D))
4. Trace: (RAC '(D))
4. Trace: RAC ==> D
3. Trace: RAC ==> D
2. Trace: RAC ==> D
1. Trace: RAC ==> D
D
[8]> ( trace rdc )
;; Tracing function RDC.
(RDC)
[9]> ( rdc '( a ) )
1. Trace: (RDC '(A))
1. Trace: RDC ==> NIL
```

```
[10]> ( rdc '( a b c d e ) )
1. Trace: (RDC '(A B C D E))
2. Trace: (RDC '(B C D E))
3. Trace: (RDC '(C D E))
4. Trace: (RDC '(D E))
5. Trace: (RDC '(E))
5. Trace: RDC ==> NIL
4. Trace: RDC ==> (D)
3. Trace: RDC ==> (C D)
2. Trace: RDC ==> (B C D)
1. Trace: RDC ==> (A B C D)
(A B C D)
[11]> ( untrace rac )
(RAC)
[12]> ( untrace rdc )
(RDC)
[13]> ( trace snoc )
;; Tracing function SNOC.
(SNOC)
[14]> ( snoc BLUE '() )

*** - SYSTEM::READ-EVAL-PRINT: variable BLUE has no
      value
The following restarts are available:
USE-VALUE      :R1      Input a value to be used instead o
f BLUE.
STORE-VALUE    :R2      Input a new value for BLUE.
ABORT          :R3      Abort main loop
Break 1 [15]> ( snoc 'blue '() )
1. Trace: (SNOC 'BLUE 'NIL)
1. Trace: SNOC ==> (BLUE)
(BLUE)
```

```
Break 1 [15]> ( snoc 'BLUE '( RED ) )
1. Trace: (SNOC 'BLUE '(RED))
2. Trace: (SNOC 'BLUE 'NIL)
2. Trace: SNOC ==> (BLUE)
1. Trace: SNOC ==> (RED BLUE)
(RED BLUE)
Break 1 [15]> ( snoc 'BLUE '( NAVY SKY TURQUOISE TEAL ) )
1. Trace: (SNOC 'BLUE '(NAVY SKY TURQUOISE TEAL))
2. Trace: (SNOC 'BLUE '(SKY TURQUOISE TEAL))
3. Trace: (SNOC 'BLUE '(TURQUOISE TEAL))
4. Trace: (SNOC 'BLUE '(TEAL))
5. Trace: (SNOC 'BLUE 'NIL)
5. Trace: SNOC ==> (BLUE)
4. Trace: SNOC ==> (TEAL BLUE)
3. Trace: SNOC ==> (TURQUOISE TEAL BLUE)
2. Trace: SNOC ==> (SKY TURQUOISE TEAL BLUE)
1. Trace: SNOC ==> (NAVY SKY TURQUOISE TEAL BLUE)
(NAVY SKY TURQUOISE TEAL BLUE)
Break 1 [15]> ( untrace snoc )
(SNOC)
Break 1 [15]> ( trace palindrome-p )
;; Tracing function PALINDROME-P.
(PALINDROME-P)
Break 1 [15]> ( palindrome-p '() )
1. Trace: (PALINDROME-P 'NIL)
1. Trace: PALINDROME-P ==> T
T
Break 1 [15]> ( palindrome-p '( Palindrome ) )
1. Trace: (PALINDROME-P '(PALINDROME))
1. Trace: PALINDROME-P ==> T
T
```

```
Break 1 [15]> ( palindrome-p '( CLOSE SLOC ) )
1. Trace: (PALINDROME-P '(CLOSE SLOC))
1. Trace: PALINDROME-P ==> NIL
NIL
Break 1 [15]> ( palindrome-p '( FOOD DRINK FOOD ) )
1. Trace: (PALINDROME-P '(FOOD DRINK FOOD))
2. Trace: (PALINDROME-P '(DRINK))
2. Trace: PALINDROME-P ==> T
1. Trace: PALINDROME-P ==> T
T
Break 1 [15]> ( palindrome-p '( 1 2 3 4 5 4 2 3 1 ) )
1. Trace: (PALINDROME-P '(1 2 3 4 5 4 2 3 1))
2. Trace: (PALINDROME-P '(2 3 4 5 4 2 3))
2. Trace: PALINDROME-P ==> NIL
1. Trace: PALINDROME-P ==> NIL
NIL
Break 1 [15]> ( palindrome-p '( HEY HEY MY MY MY MY HEY HE
Y ) )
1. Trace: (PALINDROME-P '(HEY HEY MY MY MY MY HEY HEY))
2. Trace: (PALINDROME-P '(HEY MY MY MY MY HEY))
3. Trace: (PALINDROME-P '(MY MY MY MY))
4. Trace: (PALINDROME-P '(MY MY))
5. Trace: (PALINDROME-P 'NIL)
5. Trace: PALINDROME-P ==> T
4. Trace: PALINDROME-P ==> T
3. Trace: PALINDROME-P ==> T
2. Trace: PALINDROME-P ==> T
1. Trace: PALINDROME-P ==> T
T
```

Task 2:

Task 2 Code:

```
( defun select ( n l )
   ( cond
      ( (> n 0)
        ( select (- n 1) (cdr l) )
      )
      ( t
        ( car l )
      )
   )
)

( defun pick ( l )
   ( select ( random ( length l ) ) l )
)
```

Task 2 Demo:

```
[1]> ( load "lp.l" )
;; Loading file lp.l ...
;; Loaded file lp.l
T
[2]> ( trace select )
;; Tracing function SELECT.
(SELECT)
[3]> ( select 4 '( 1 2 3 4 5 6 7 ) )
1. Trace: (SELECT '4 '(1 2 3 4 5 6 7))
2. Trace: (SELECT '3 '(2 3 4 5 6 7))
3. Trace: (SELECT '2 '(3 4 5 6 7))
4. Trace: (SELECT '1 '(4 5 6 7))
5. Trace: (SELECT '0 '(5 6 7))
5. Trace: SELECT ==> 5
4. Trace: SELECT ==> 5
3. Trace: SELECT ==> 5
2. Trace: SELECT ==> 5
1. Trace: SELECT ==> 5
5
[4]> ( select 2 '( 1 2 3 ) )
1. Trace: (SELECT '2 '(1 2 3))
2. Trace: (SELECT '1 '(2 3))
3. Trace: (SELECT '0 '(3))
3. Trace: SELECT ==> 3
2. Trace: SELECT ==> 3
1. Trace: SELECT ==> 3
3
[5]> ( select 0 '( 0 1 2 3 4 ) )
1. Trace: (SELECT '0 '(0 1 2 3 4))
1. Trace: SELECT ==> 0
0
[6]> ( untrace select )
(SELECT)
```

```
[7]> ( pick '(0 1 2 3 4 5 ) )
5
[8]> ( pick '(5 4 3 2 1 0 ) )
5
[9]> ( pick '(0 1 2 3 4 ) )
0
[10]> ( pick '(9 8 7 6 ) )
6
```

Task 3:

Task 3 Code:

```
( defun sum ( l )
   ( cond
      ( ( null l )
        0
      )
      ( t
        ( + ( car l ) ( sum ( cdr l ) ) )
      )
   )
)

( defun product ( l )
   ( cond
      ( ( null l )
        1
      )
      ( t
        ( * ( car l ) ( product ( cdr l ) ) )
      )
   )
)
```

Task 3 Demo:

```
[2]> ( trace sum )
;; Tracing function SUM.
(SUM)
[3]> ( sum '() )
1. Trace: (SUM 'NIL)
1. Trace: SUM ==> 0
0
[4]> ( sum '( 496 )
)
1. Trace: (SUM '(496))
2. Trace: (SUM 'NIL)
2. Trace: SUM ==> 0
1. Trace: SUM ==> 496
496
[5]> ( sum '( 1 11 111 ) )
1. Trace: (SUM '(1 11 111))
2. Trace: (SUM '(11 111))
3. Trace: (SUM '(111))
4. Trace: (SUM 'NIL)
4. Trace: SUM ==> 0
3. Trace: SUM ==> 111
2. Trace: SUM ==> 122
1. Trace: SUM ==> 123
123
[6]> ( sum '( 1 2 3 4 5 6 7 8 9 10 ) )
1. Trace: (SUM '(1 2 3 4 5 6 7 8 9 10))
2. Trace: (SUM '(2 3 4 5 6 7 8 9 10))
3. Trace: (SUM '(3 4 5 6 7 8 9 10))
4. Trace: (SUM '(4 5 6 7 8 9 10))
5. Trace: (SUM '(5 6 7 8 9 10))
6. Trace: (SUM '(6 7 8 9 10))
7. Trace: (SUM '(7 8 9 10))
8. Trace: (SUM '(8 9 10))
9. Trace: (SUM '(9 10))
10. Trace: (SUM '(10))
11. Trace: (SUM 'NIL)
11. Trace: SUM ==> 0
10. Trace: SUM ==> 10
9. Trace: SUM ==> 19
8. Trace: SUM ==> 27
7. Trace: SUM ==> 34
6. Trace: SUM ==> 40
5. Trace: SUM ==> 45
4. Trace: SUM ==> 49
3. Trace: SUM ==> 52
2. Trace: SUM ==> 54
1. Trace: SUM ==> 55
55
```

```
[7]> ( trace product )
;; Tracing function PRODUCT.
(PRODUCT)
[8]> ( product '() )
1. Trace: (PRODUCT 'NIL)
1. Trace: PRODUCT ==> 1
1
[9]> ( product '( 496 ) )
1. Trace: (PRODUCT '(496))
2. Trace: (PRODUCT 'NIL)
2. Trace: PRODUCT ==> 1
1. Trace: PRODUCT ==> 496
496
[10]> ( product '( 1 11 111 ) )
1. Trace: (PRODUCT '(1 11 111))
2. Trace: (PRODUCT '(11 111))
3. Trace: (PRODUCT '(111))
4. Trace: (PRODUCT 'NIL)
4. Trace: PRODUCT ==> 1
3. Trace: PRODUCT ==> 111
2. Trace: PRODUCT ==> 1221
1. Trace: PRODUCT ==> 1221
1221
[11]> ( product '( 1 2 3 4 5 6 7 8 9 10 ) )
1. Trace: (PRODUCT '(1 2 3 4 5 6 7 8 9 10))
2. Trace: (PRODUCT '(2 3 4 5 6 7 8 9 10))
3. Trace: (PRODUCT '(3 4 5 6 7 8 9 10))
4. Trace: (PRODUCT '(4 5 6 7 8 9 10))
5. Trace: (PRODUCT '(5 6 7 8 9 10))
6. Trace: (PRODUCT '(6 7 8 9 10))
7. Trace: (PRODUCT '(7 8 9 10))
8. Trace: (PRODUCT '(8 9 10))
9. Trace: (PRODUCT '(9 10))
10. Trace: (PRODUCT '(10))
11. Trace: (PRODUCT 'NIL)
11. Trace: PRODUCT ==> 1
10. Trace: PRODUCT ==> 10
9. Trace: PRODUCT ==> 90
8. Trace: PRODUCT ==> 720
7. Trace: PRODUCT ==> 5040
6. Trace: PRODUCT ==> 30240
5. Trace: PRODUCT ==> 151200
4. Trace: PRODUCT ==> 604800
3. Trace: PRODUCT ==> 1814400
2. Trace: PRODUCT ==> 3628800
1. Trace: PRODUCT ==> 3628800
3628800
```

Task 4:

Task 4 Code:

```
( defun iota ( n )
   ( cond
      ( ( = n 0 )
         ()
      )
      ( t
         ( snoc n ( iota ( - n 1 ) ) )
      )
   )
)

( defun duplicate ( n lo )
   ( cond
      ( ( = n 0 )
         ()
      )
      ( t
         ( snoc lo ( duplicate ( - n 1 ) lo ) )
      )
   )
)
```

Task 4 Demo:

```
[1]> ( load "lp.l" )
;; Loading file lp.l ...
;; Loaded file lp.l
T
[2]> ( trace iota )
;; Tracing function IOTA.
(IOTA)
[3]> ( iota 1 )
1. Trace: (IOTA '1)
2. Trace: (IOTA '0)
2. Trace: IOTA ==> NIL
1. Trace: IOTA ==> (1)
(1)
[4]> ( iota 10 )
1. Trace: (IOTA '10)
2. Trace: (IOTA '9)
3. Trace: (IOTA '8)
4. Trace: (IOTA '7)
5. Trace: (IOTA '6)
6. Trace: (IOTA '5)
7. Trace: (IOTA '4)
8. Trace: (IOTA '3)
9. Trace: (IOTA '2)
10. Trace: (IOTA '1)
11. Trace: (IOTA '0)
11. Trace: IOTA ==> NIL
10. Trace: IOTA ==> (1)
9. Trace: IOTA ==> (1 2)
8. Trace: IOTA ==> (1 2 3)
7. Trace: IOTA ==> (1 2 3 4)
6. Trace: IOTA ==> (1 2 3 4 5)
5. Trace: IOTA ==> (1 2 3 4 5 6)
4. Trace: IOTA ==> (1 2 3 4 5 6 7)
3. Trace: IOTA ==> (1 2 3 4 5 6 7 8)
2. Trace: IOTA ==> (1 2 3 4 5 6 7 8 9)
1. Trace: IOTA ==> (1 2 3 4 5 6 7 8 9 10)
(1 2 3 4 5 6 7 8 9 10)
```

```
[5]> ( trace duplicate )
;; Tracing function DUPLICATE.
(DUPLICATE)
[6]> ( duplicate 3 'boing )
1. Trace: (DUPLICATE '3 'BOING)
2. Trace: (DUPLICATE '2 'BOING)
3. Trace: (DUPLICATE '1 'BOING)
4. Trace: (DUPLICATE '0 'BOING)
4. Trace: DUPLICATE ==> NIL
3. Trace: DUPLICATE ==> (BOING)
2. Trace: DUPLICATE ==> (BOING BOING)
1. Trace: DUPLICATE ==> (BOING BOING BOING)
(BOING BOING BOING)
[7]> ( duplicate 9 '(9) )
1. Trace: (DUPLICATE '9 '(9))
2. Trace: (DUPLICATE '8 '(9))
3. Trace: (DUPLICATE '7 '(9))
4. Trace: (DUPLICATE '6 '(9))
5. Trace: (DUPLICATE '5 '(9))
6. Trace: (DUPLICATE '4 '(9))
7. Trace: (DUPLICATE '3 '(9))
8. Trace: (DUPLICATE '2 '(9))
9. Trace: (DUPLICATE '1 '(9))
10. Trace: (DUPLICATE '0 '(9))
10. Trace: DUPLICATE ==> NIL
9. Trace: DUPLICATE ==> ((9))
8. Trace: DUPLICATE ==> ((9) (9))
7. Trace: DUPLICATE ==> ((9) (9) (9))
6. Trace: DUPLICATE ==> ((9) (9) (9) (9))
5. Trace: DUPLICATE ==> ((9) (9) (9) (9) (9))
4. Trace: DUPLICATE ==> ((9) (9) (9) (9) (9) (9))
3. Trace: DUPLICATE ==> ((9) (9) (9) (9) (9) (9) (9))
2. Trace: DUPLICATE ==> ((9) (9) (9) (9) (9) (9) (9) (9))
1. Trace: DUPLICATE ==> ((9) (9) (9) (9) (9) (9) (9) (9) (9))
((9) (9) (9) (9) (9) (9) (9) (9) (9))
```

Task 5:

Task 5 Code:

```
( defun factorial ( n )
   ( product ( iota n ) )
)

( defun power ( n pow )
   ( product ( duplicate n pow ) )
)
```

Task 5 Demo:

```
[1]> ( load "lp.l" )
;; Loading file lp.l ...
;; Loaded file lp.l
T
[2]> ( factorial 5 )
120
[3]> ( factorial 10 )
3628800
[4]> ( power 5 6 )
7776
[5]> ( power 2 16 )
256
```

Task 6:

Task 6 Code:

```
( defun filter-in ( p li )
   ( cond
      ( ( null li) () )
      ( ( funcall p ( car li ) )
         ( cons ( car li ) (filter-in p ( cdr li ) ) )
      )
      ( t
         ( filter-in p ( cdr li ) )
      )
   )
)

( defun filter-out ( p li )
   ( cond
      ( ( null li) () )
      ( ( funcall p ( car li ) )
         ( filter-out p ( cdr li ) )
      )
      ( t
         ( cons (car li) ( filter-out p ( cdr li ) ) )
      )
   )
)

( defun single-digit ( n )
   ( < n 10 )
)
```

Task 6 Demo:

```
[2]> ( filter-in #'palindrome-p '( ( house mouse ) ( ialai is ialai ) ( 1 2 3 2 1 ) (oneword) ) )
((IALAI IS IALAI) (1 2 3 2 1) (ONEWORD))
[3]> ( filter-in #'singleton-p '( ( hi ) ( is it ) ( im ) ( hand towel ) ( a ) ( small bowl ) ( cat ) ) )
((HI) (IM) (A) (CAT))
[4]> ( filter-in #'single-digit '( 100 5 4 21 2 ) )
(5 4 2)
[5]> ( filter-out #'palindrome-p '( ( house mouse ) ( ialai is ialai ) ( 1 2 3 2 1 ) (oneword) ) )
((HOUSE MOUSE))
[6]> ( filter-out #'singleton-p '( ( hi ) ( is it ) ( im ) ( hand towel ) ( a ) ( small bowl ) ( cat ) ) )
((IS IT) (HAND TOWEL) (SMALL BOWL))
[7]> ( filter-out #'single-digit '( 100 5 4 21 2 ) )
(100 21)
```

Task 7:

Task 7 Code:

```
( defun take-from ( obj li )
   ( cond
      ( (null li) () )
      ( ( equal obj ( car li ) )
           ( take-from obj ( cdr li ) )
      )
      ( t
           ( cons (car li) ( take-from obj ( cdr li ) ) )
      )
   )
)
```

Task 7 Demo:

```
[2]> ( take-from 'banana '( peach pear banana apple orange ) )
(PEACH PEAR APPLE ORANGE)
[3]> ( take-from '10 '( 1 10 20 30 40 30 20 10 1 ) )
(1 20 30 40 30 20 1)
[4]> ( take-from 'high '( high low road ) )
(LOW ROAD)
```

Task 8:

Task 8 Code:

```
( defun random-permutation (li)
   ( cond
      ( (null li) li)
      (t
         ( setf element ( pick li ) )
         ( setf li ( remove element li :count 1) )
         ( snoc element ( random-permutation li ) )
      )
   )
)
```

Task 8 Demo:

```
[2]> ( random-permutation '( this used to be a legible sentence ) )
(THIS USED LEGIBLE TO SENTENCE BE A)
[3]> ( random-permutation '( strong the force is with this one ) )
(ONE THIS IS STRONG WITH FORCE THE)
[4]> ( random-permutation '( 1 2 3 4 5 6 7 8 9 10 11 12 ) )
(9 8 6 12 5 4 3 7 2 10 1 11)
[5]> ( trace random-permutation )
;; Tracing function RANDOM-PERMUTATION.
(RANDOM-PERMUTATION)
[6]> ( random-permutation '( how random is this ) )
1. Trace: (RANDOM-PERMUTATION '(HOW RANDOM IS THIS))
2. Trace: (RANDOM-PERMUTATION '(HOW IS THIS))
3. Trace: (RANDOM-PERMUTATION '(IS THIS))
4. Trace: (RANDOM-PERMUTATION '(THIS))
5. Trace: (RANDOM-PERMUTATION 'NIL)
5. Trace: RANDOM-PERMUTATION ==> NIL
4. Trace: RANDOM-PERMUTATION ==> (THIS)
3. Trace: RANDOM-PERMUTATION ==> (THIS IS)
2. Trace: RANDOM-PERMUTATION ==> (THIS IS HOW)
1. Trace: RANDOM-PERMUTATION ==> (THIS IS HOW RANDOM)
(THIS IS HOW RANDOM)
```

Task 9:

Task 9 Demo:

```
[2]> ( mapcar #'car '( ( a b c ) ( d e ) ( f g h i ) ) )
(A D F)
[3]> ( mapcar #'cons '( a b c ) '( x y z ) )
((A . X) (B . Y) (C . Z))
[4]> ( mapcar #'* '( 1 2 3 4 ) '( 4 3 2 1 ) '( 1 10 100 1000 ) )
(4 60 600 4000)
[5]> ( mapcar #'cons '( a b c ) '( ( one ) ( two ) ( three ) ) )
((A ONE) (B TWO) (C THREE))
[6]> ( mapcan #'cons '( a b c ) '( ( one ) ( two ) ( three ) ) )
(A ONE B TWO C THREE)
```

Task 10:

Task 10 Demo:

```
[2]> ( mapcar #'expt '( 2 2 2 2 2 ) '( 0 1 2 3 4 ) )
(1 2 4 8 16)
[3]> ( mapcar #'cadr '( ( a b c ) ( d e f ) ( g h i ) ( k j l ) ) )
(B E H J)
[4]> ( mapcar #'pick '( ( big small ) ( red yellow blue green ) ( machine moon book ) ) )
(SMALL YELLOW MOON)
[5]> ( mapcar #'cons ( iota 4 ) ( duplicate 4 'and ) )
((1 . AND) (2 . AND) (3 . AND) (4 . AND))
[6]> ( mapcar #'iota ( iota 4 ) )
((1) (1 2) (1 2 3) (1 2 3 4))
[7]> ( mapcan #'iota ( iota 4 ) )
(1 1 2 1 2 3 1 2 3 4)
```

Task 11:

Task 11 Code:

```
( defun replace-lcr ( loc e li )
   ( cond
      ( ( equal loc 'left ) ( cons e ( cdr li ) ) )
      ( ( equal loc 'center) ( list ( first li ) e ( third li ) ) )
      ( ( equal loc 'right ) ( list ( first li ) ( second li ) e ) )
   )
)

( defun uniform-p ( li )
   ( cond
      ( ( null li ) t )
      ( ( = ( length li ) 1 ) t )
      ( ( equal ( car li ) ( second li ) ) ( uniform-p ( cdr li) ) )
      ( t
        nil )
   )
)

( defun flush-p ( li )
   ( uniform-p ( mapcar #'cdr li ) )
)
```

Task 11 Demo:

```
[1]> ( load "ditties.l" )
;; Loading file ditties.l ...
;; Loaded file ditties.l
T
[2]> ( replace-lcr 'left 't '( h h h ) )
(T H H)
[3]> ( replace-lcr 'center 't '( h h h ) )
(H T H)
[4]> ( replace-lcr 'right 't '( h h h ) )
(H H T)
[5]> ( uniform-p '( this is not uniform ) )
NIL
[6]> ( uniform-p '( uniform uniform uniform ) )
T
[7]> ( flush-p '( ( king . hearts ) ( ace . hearts ) ) )
T
[8]> ( flush-p '( ( king . spades ) ( ace . hearts ) ) )
NIL
```